Applying RAMS Principles to the Development of a Safety-Critical Java Specification

Dr. Kelvin Nilsen
Aonix

Almost all software for aerospace and defense applications is required to satisfy reliable, available, maintainable, and safe (RAMS) objectives. While many RAMS issues are best addressed by requiring that software developers consistently adhere to particular development methodologies, a development team's selection of commercial off-the-shelf technologies, including choice of programming language, run-time environment, and libraries, may also impact the team's ability to satisfy RAMS requirements. This article evaluates a proposed specification for safety-critical Java in terms of RAMS principles, comparing the use of the draft safety-critical Java standard with traditional approaches based on C, and motivating the restrictions imposed by the safety-critical Java specification in comparison with use of traditional Java and the general purpose Real-Time Specification for Java. The RAMS solutions that have been designed for the proposed safety-critical Java specification apply equally well to a breadth of defense and aerospace application domains, including hard real-time mission-critical code for communication, sensing, guidance, and automation subsystems.

The Radio Technical Commission for Aeronautics (RTCA) DO-178B guidelines [1] are designed to span a range of criticality levels. The most life-critical software components in an avionics system are characterized as Level-A. Failure of a Level-A software component is considered *catastrophic*. Without this component, further flight and/or landing of an aircraft is considered impossible. Failure of a Level-C component is considered *major*, reducing the ability of a crew to cope with flight responsibilities, but not significantly increasing the risk of a crash. The safety-critical Java specification that is under development is designed to span the full range of DO-178B levels. In this regard, it addresses both life-critical and mission-critical systems.

Satisfying DO-178B certification requirements involves considerable engineering discipline. Enforcing this discipline is the responsibility of project managers and team leaders. Peer reviews are required at every step of the development process. Extensive documentation and accountability audit trails are required to ensure that no corners are cut in design, development, and testing of the safety-critical software. The DO-178B guidelines are independent of programming language choice. Regardless of programming language, engineers are required to address all of the same issues and gather all of the same documentation artifacts. You might reasonably ask, "What difference does the choice of programming language make?"

To answer this question, it is necessary to look more closely at some of the issues that must be addressed by developers of mission-critical and safety-critical systems. This article focuses on the programming language impact with respect to four broad issues: reliability, availability, maintainability, and safety (RAMS). In discussing these issues, we draw comparisons between using C, C++, the Real-Time Specification for Java (RTSJ) [2], traditional Java [3], and the proposed safety-critical specification for Java [4, 5]. Even though these issues can be addressed satisfactorily in a number of different languages, certain languages require less effort to address than others.

The draft safety-critical specification that is discussed here is currently under consideration for submission as a Java Community Process (JCP) standard. A prototype implementation of this specification is currently under development. Based on feedback from early evaluators, we expect to make appropriate refinements to the specification before submitting the final result for standardization under the JCP. We expect submission of the standard to take place during 2006. To track the progress of this ongoing specification and standardization work, refer to <http://research.aonix.com/jsc>.

Reliability

Among key considerations of developers focused on delivering high reliability are the following:

- The language and run-time environment must be sufficiently simple so they are easily understood.
- Ideally, the language and standard libraries behave consistently across platforms. Otherwise, programmers are likely to overlook incompatibilities when shifting their development efforts or migrating software components from one platform to the next.
- To help programmers manage complexity reliably, good programming languages support abstractions that allow programmers to separate concerns between independent components.
- Over the years, computer scientists have experimented with a variety of programming language features. Some very powerful features are easily misused, with far-reaching consequences. Certain features – such as implicit coercion of integer to Boolean – have been shown to be very error-prone. Programming languages that omit dangerous and error-prone features make it easier for developers to build reliable systems.

One of the main reasons Java has been such a popular alternative to C and C++ is because it is a much more portable programming language. This portability has resulted in a variety of important benefits. First, the standard libraries behave similarly across a wide variety of central processor unit (CPU) architectures and operating systems. Second, third-party developers of open-source and commercial off-the-shelf Java components are able to distribute these libraries in portable binary representations, without regard for on which platform they will be used. Third, developers of embedded systems can develop and test their software on fast, large-memory desktop machines, and then deploy the completed software on much slower, memory-limited embedded targets without concern that the code will behave differently in the embedded environment. Fourth, the *learning curve* for programmers consists only of learning the portable platform. No additional effort is required to learn the peculiarities of each implementation of the platform running on each different real-time operating system (RTOS).

These portability benefits are relevant to developers of safety-critical code. The

relevance of these benefits to system reliability is that (1) programmers are less likely to introduce errors because they have misunderstood or overlooked peculiarities of a particular Java implementation, and (2) the ability to reuse components across different platforms means a typical safety-critical deployment has a higher percentage of mature, time-proven software components in place versus custom-tailored software components that have never been used before.

Though the Java platform is portable with respect to functional behavior, standard-edition Java does not provide portability with respect to real-time issues. For example, the amount of memory allocated to create a particular data structure may vary significantly from one Java implementation to the next. And the scheduling of threads is also highly platform-dependent. To address these issues, the draft safety-critical Java specification carefully defines the precise semantics of a very small subset of the full Java Standard Edition libraries in combination with a small subset of the full RTSJ.

The selection of these libraries focuses on providing the minimal functionality required as a portable and extensible foundation upon which to build safety-critical systems. We annotate this set of libraries to characterize which services must be time- and memory-bounded, and we make these same annotations available to application developers so they can document their intentions with respect to the code they develop. A special safety-critical byte-code verifier (static analysis tool) enforces that method implementations are entirely consistent with the supplied programmer annotations. All of this clarifies which components can be reliably used in hard real-time contexts, including interrupt handlers. Further, determination of the memory and CPU time resources required for reliable operation of hard real-time software components is automatic.

Contrast this with the typical approach of a C or C++ developer. Since these languages were not designed for multi-threaded environments, the existing standards do not address the code generation issues that affect information sharing between threads. If a particular thread modifies a shared variable, even a variable that is defined as volatile, the propagation of the new value to other threads that are monitoring the same variable is highly nonportable. C and C++ programmers must understand the code generation model for each of the optimization modes they choose to use with their compiler. They must understand the underlying architecture's cache coherency model, and must study the underlying, real-time operating system's thread-scheduling semantics.

Often, the information required to develop reliable code is not well documented, and programmers have to spend considerable time and effort performing detective work to make sure they fully understand the platform they are targeting. This *detective work* often consists of trial-and-error experimentation. This may leave developers with lingering uncertainties as to whether their experimentation has uncovered all of the underlying platform's peculiarities and that they fully understand them. The software they write must be tested extensively to make sure it runs correctly on the targeted platform; however, the assumptions on which the correct operation of the software depends are rarely documented. If this software is ever moved to a different CPU, compiler, or RTOS, then extensive code review and retesting are required.

"Often, the information required to develop reliable code is not well documented, and programmers have to spend considerable time and effort performing detective work to make sure they fully understand the platform they are targeting."

The RTSJ exhibits some of the same difficulties encountered by C and C++ developers. Though this specification more carefully constrains real-time operation of threads than Standard Edition Java, it does not fully specify the semantics of the real-time libraries. Many of the capabilities offered within the RTSJ framework are optional, and the exact semantics of certain other features such as precisely when to trigger execution of a deadline-overrun handler are incompletely defined. This is one of the reasons that the draft safety-critical Java specification selects a subset of the full RTSJ framework. This subset specifically excludes capabilities that are difficult to define and implement in a portable way, and avoids many complex and costly features that are less relevant to developers of safety-critical or hard real-time systems. This results in a much smaller, more easily understood subset of core functionality, early implementations of which run more than three times faster than existing full RTSJ implementations on certain benchmarks.

Certain programming abstractions that are critical to developers of safety-critical code are totally irrelevant to typical developers of management information systems. Thus, languages like C, C++, and Standard Edition Java do not provide support for these abstractions. Consider, for example, the need of safety-critical developers to know the following:

1. Exactly how much real memory is required for the run-time stack of a safety-critical thread. (Note that safety-critical systems generally do not have hard disks and have no support for virtual memory or for dynamic expansion of a run-time stack.)
2. Exactly how much memory is required to represent a particular data structure that is going to be shared between multiple threads.
3. Exactly how much CPU time is required to reliably execute particular threads within real-time timing constraints [6].
4. Exactly how much time each thread might need to block while waiting for access to a shared resource that is required to complete a particular safety-critical task on schedule [6].

The safety-critical Java proposal addresses these issues by introducing standard meta-data annotations that allow programmers to constrain the behavior of particular methods. For example, an @StaticAnalyzable annotation denotes that the implementation of the method must adhere to particular style guidelines that allow the memory usage and the CPU time to be automatically determined. A special safety-critical byte-code verifier enforces that the code conforms to these style guidelines, and a separate static analysis tool determines the resource needs for each targeted platform.

Another special, hard real-time abstraction supported by the draft safety-critical Java proposal is a special synchronization mechanism known as priority ceiling emulation. With this mechanism, the programmer is required to specify an upper bound on the priorities of threads that might attempt to perform synchronized access to each lock. This upper bound is known as the ceiling priority.

When a thread acquires this lock, the thread's priority is immediately boosted to the ceiling priority. When the thread releases the lock, the thread's priority is restored to its original value.

Within the safety-critical profile, priority ceiling emulation is the only supported synchronization mechanism. A specialization of priority ceiling locks is known within the safety-critical profile as atomic locking. Programmers make use of a special syntax to identify objects that use atomic priority ceiling emulation to coordinate shared access between multiple threads. For each such object, the safety-critical byte-code verifier assures that the component does not perform any blocking operation while a given thread holds the object's atomic lock. With this byte-code enforcement in place, the implementation of atomic locks (on single-processor systems[1]) is very efficient and the worst-case blocking time to access an atomic lock is easily analyzed.

In particular, if a given thread is able to reach the point of entry to that lock, it is guaranteed that no other thread owns the lock. If another thread owned the lock, it would be executing at a higher priority so this thread would not be running. Thus, the blocking time is always zero. This important abstraction is recommended for all resource sharing among hard real-time safety-critical threads. It can only be implemented reliably through coordination between static analysis tools and run-time services. This coordination is provided within the safety-critical Java profile. It is not available in C, C++, Standard Edition Java, or the RTSJ.

Another important consideration is management of temporary scratch pad memory during the execution of hard, real-time, safety-critical threads. Temporary memory may be required to support digital signal processor analysis of sensor inputs, and to manage buffers for communication with redundant onboard, safety-critical modules and with remote systems that are, for example, providing air traffic control directives. The C malloc()/free() and C++ new()/delete() services are subject to memory fragmentation. Thus, they should not be used in memory-limited, safety-critical systems.

Java's automatic garbage collection system can defragment the dynamic memory heap. But the complexity and asynchrony of automatic garbage collection are very difficult to certify to the satisfaction of Federal Aviation Administration auditors. For this reason, the draft safety-critical Java profile provides an alternative memory management technique. We identify this approach as safe-scoped memory. It is a generalization of the RTSJ's scoped memory abstraction. In essence, the draft safety-critical profile allows objects to be allocated within the activation frames of each method.

In contrast with C and C++, which also allow objects to be allocated on the run-time stack, the safety-critical Java proposal uses programmer annotations to track the flow of stack-allocated objects; its byte-code verifier guarantees that no reference to a stack-allocated object lives longer than the object itself. This solves the all-too-common dangling pointer problem that plagues C and C++ development.

In contrast with the full-RTSJ scoped memory abstractions, which also prevent dangling pointers, the safety-critical profile detects scoped memory violations at compile time rather than at run-time. In summary, the proposed safety-critical Java temporary memory abstractions support much more reliable operations than common alternatives.

"Certain programming abstractions that are critical to developers of safety-critical code are totally irrelevant to typical developers of management information systems."

Availability

Availability addresses the requirement that high-integrity software must be always ready to perform its function. Availability is often measured in terms of a quantity of nines, representing the percentage of total time that the high-integrity system can be relied upon to fulfill its duty. For example, *seven nines* availability means the system is running reliably 99.99999 percent of the time. Note that seven nines operation allows only half a second of downtime per year.

Strategies for assuring high availability generally consist of a combination of the following:

- Take every reasonable action to maximize reliability as this will extend the *mean time between failures*. Reliability was discussed in the previous section.
- Minimize downtime whenever failures are encountered by doing the following:
  - Provide fast, deterministic restart of a failed system.
  - Provide fast, deterministic reconfiguration of software device drivers whenever failed hardware components might have to be replaced with upgraded hardware – if possible, upgrade device drivers and hot-swap hardware without rebooting.
- Support redundant computation and communication so that standby components can quickly take responsibility for ongoing services when particular components fail.

Providing a fast, deterministic restart of a failed system is an obvious requirement for high-availability applications. Achieving this objective is not trivial. Consider how long it takes to turn on typical computers and various smart gadgets such as cell phones. Startup is especially troublesome in typical Java environments, including compliant, full RTSJ implementations, because the startup process includes dynamic loading of byte code and translation of this byte code into native machine language by just-in-time (JIT) compilers.

The draft safety-critical Java profile addresses this concern by requiring static compilation, initialization, and linking of components. Unlike traditional Java, in which the initial values of many shared variables – even of so-called constant variables – depends on the order in which certain non-deterministic startup activities are performed, the safety-critical profile's byte-code verifier enforces fully deterministic initialization of shared static variables. The safety-critical Java linker binds all of the components together and initializes shared memory in the static load image. The large majority of this load image can be burned into read-only memory (ROM) and accessed directly out of ROM. Only objects with variable contents must be copied into random access memory at startup time.

Occasionally, highly available systems experience hardware failures. When hardware must be replaced, it is often necessary to replace software device drivers that control the hardware. Few real-time operating systems provide direct support for dynamic replacement of device drivers. Larger desktop operating systems usually support plug-and-play devices, but the protocols for using plug-and-play technologies are not especially reliable. Often, conflicts between device drivers supplied

by different vendors result in unreliable operation of the newly configured environment. A goal for the safety-critical Java profile is to support reliable and deterministic reconfiguration of device drivers, both for situations in which the device drivers are replaced without downtime and for situations in which hardware replacement requires system reboot.

As with many other issues, the safety-critical Java profile tackles this challenge using a combination of programmer annotations, special byte-code verification, and reliable run-time memory management services, specifically the following:

1. All of the memory consumed by a device driver is organized as a contiguous region of a budgeted size. If a particular device driver is removed, all of the memory previously set aside for that device driver is instantly reclaimed without any fragmentation issues. This memory can serve the needs of the replacement device driver.
2. The safety-critical Java profile provides annotations to allow programmers to describe the interface requirements of device drivers in sufficient detail to allow the static analysis tools to verify that a particular device driver is a suitable replacement for an existing device driver. Specifically, programmers can do the following:
   a. Characterize which inside/outside ports the device needs to read and write.
   b. Identify to which interrupt vectors the device driver needs to respond.
   c. Specify the maximum amount of time the device driver is allowed to hold particular interrupts disabled.
   d. Define the precise entry points whereby application code communicates with the device driver, and enforce that every invocation of these services is consistent with the interface expectations for that service.

Support for redundant computations and failover processing is not directly supported by the safety-critical Java profile. It is important to note that the Java platform was originally introduced as an Internet programming language. As such, there is considerable experience using Java for networked applications. Since the draft safety-critical Java profile establishes a strong foundation for reuse of portable, hard, real-time software components, it should be straightforward to develop portable libraries to support safety-critical, networked communications to support fault-tolerant and high-availability redundant computations.

Maintainability

With many safety- and mission-critical systems, fielded software must endure for many years, often several decades. During its useful lifetime, this software evolves in response to changing platform requirements, new communication protocols, integration of new functionality, and so forth. Over the lifetime of a particular system, it is common for the costs of software maintenance to far exceed the costs of the original software development. Typical maintenance activities include (1) fixing a bug, (2) addressing a performance issue, or (3) adding new functionality.

Maintaining real-time software is particularly difficult because the declared interfaces for C and C++ components do not reflect all the conditions required for reliable composition of real-time components. This means developers who are called upon to make changes to existing software cannot determine by looking at the component interface alone what rules they must follow for their changes to integrate reliably with other existing software. In particular, they do not know the following:

1. Whether incoming arguments might refer to temporary objects or permanent objects, and whether the referenced objects might be shared with other threads.
2. Whether memory resources have been budgeted to allow the implementation of a particular service to allocate permanent or temporary objects.
3. Which memory allocation budgets must be increased for this revised component to be able to reliably allocate additional memory.
4. Which memory allocation budgets can be decreased to make this component run more efficiently.
5. Which task cost-estimates must be modified if changes to this component affect its CPU time requirements.

"A goal for the safety-critical Java profile is to support reliable and deterministic reconfiguration of device drivers, both for situations in which the device drivers are replaced without downtime, and for situations in which hardware replacement requires system reboot."

Maintainers of real-time software must search for all the contexts in which particular components *reside* to determine what sort of changes they may make to those components without compromising system reliability.

Scalability is a generalization of maintainability. Many modern software systems experience evolutionary change that tracks Moore's Law [7]. As processors and computer memory decrease in cost and increase in capacity, software grows in size and complexity to match the new capacity. Studies of certain commercial, embedded software systems have observed that it is common for software size to double every 18 to 24 months [8].

Compared with C and C++, Java has shown tremendous strengths as a platform to support easy integration and economic scalability. This is because all of the Java software is very portable, and because strong object-oriented abstractions mean that independently developed components integrate cleanly, without compromising the integrity of each other's encapsulation boundaries. C, in contrast, offers very little to help manage the complexity of ever-expanding software systems. With its object-oriented features, C++ does much better than C at separating concerns of independent software development teams to facilitate software maintenance and scalability issues. However, the lack of true portability, the inherent complexity in the language itself, and its lack of automatic garbage collection makes C++ a more difficult tool than Java in its support for software maintenance and scalability.

The proposed safety-critical Java profile addresses these issues by doing the following:

1. Maintaining real-time software as *Vanilla Java* source code with Java Development Kit 5.0-style meta-data annotations to document the interface requirements associated with each software component.
2. Providing automatic consistency checking between independent interfaces, assuring that each method invocation satisfies the annotated interface requirements, that overriding method interfaces are consistent with the over-

ridden interfaces, and that method implementations are entirely consistent with the annotated method interface declarations.
3. Providing automated analysis to determine memory and CPU-time resource requirements to allow automatic configuration of resource budgeting and real-time scheduling each time any system component is modified.

Tools to automate the required consistency checking and resource needs analysis are not generally available for C and C++ development.

Safety

With respect to software systems, *safety* represents the notion that the computer system will do no harm. In this regard, safety is opposed to availability and reliability. The safest computer system might be the system that never gets powered on. Assuming that we are required to satisfy safety objectives in combination with reliability and availability objectives, the safety consideration consists primarily of satisfying safety certification requirements.

Of particular relevance to safety certification requirements is the mechanism for deployment of native machine code. Level-A certification requires that all code coverage analysis and testing be performed with the native machine language, and that responsibility for every machine code instruction and for every test case be traceable from original system requirements to architecture and design, to source code and test plans, and finally to machine code.

If certain machine instructions are not exercised sufficiently by the existing test cases, developers are required to analyze whether the code is really necessary to satisfy the system requirements. If that code is not necessary, the corresponding source code should be removed or restructured to make it more consistent with the system requirements. If the code is necessary, the test plan must be modified to make the test plan consistent with the original system requirements. In some cases, failure of test cases to cover all machine code reveals inaccuracies or inconsistencies in the original system requirements. In that situation, the original system requirements must be refined. Always, a complete traceability audit trail must be maintained.

Note that the traditional Java execution model is entirely inconsistent with this requirement for traceability from source code to machine code. Traditional Java virtual machines hide the translation of byte code to machine code within a JIT compiler that is part of the run-time environment. Some sophisticated virtual machines actually produce multiple native-code translations for each byte-code method, optimizing the code differently in each translation based on run-time profiling information. The safety-critical Java profile addresses this issue by supporting deterministic compilation, linking, and initialization model. The entire safety-critical application is translated to native machine code and linked into a ROM-loadable image prior to execution. Since this technology is designed to support safety-critical development, tools will facilitate mappings between machine code and the corresponding source code.

C and C++ development tool chains provide similar traceability support between source code and machine code.

Summary

Across the spectrum of RAMS objectives, the draft safety-critical Java specification offers important benefits over alternative approaches based on C, C++, traditional Java, or RTSJ Java. As commercial implementations of the proposed safety-critical Java standard become available, developers of safety-critical systems will be able to more economically deliver high-quality software that satisfies RAMS objectives.◆

References

1. Radio Technical Commission for Aeronautics, Inc. "RTCA DO-178B, Software Considerations in Airborne Systems and Equipment Certification." Washington, D.C.: RTCA, 1 Dec. 1992 <www.rtca.org>.
2. Bollella, G., J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. The Real-Time Specification for Java. Addison-Wesley, Jan. 2000 <www.rtj.org>.
3. Sun Microsystems Inc. "The Java Language Environment: A White Paper." Mountain View, CA: Sun Microsystems, Inc., 1995 <http://java.sun.com/docs/white/langenv>.
4. Nilsen, K. "Making Effective Use of the Real-Time Specification for Java." San Diego, CA: Aonix, Oct. 2004 <http://research.aonix.com/jsc/rtsj.issues.9-04.pdf>.
5. Nilsen, K. "Draft Guidelines for Scalable Java Development of Real-Time Systems." San Diego, CA: Aonix, May 2005 <http://research.aonix.com/jsc/rtjava.guidelines.5-6-05.pdf>.
6. Klein, M., T. Ralya, B. Pollak, and R. Obenza. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, Nov. 1993 <www.sei.cmu.edu/publications/books/other-books/rma.hndbk.html>.
7. Moore, Gordon. "Cramming More Components Onto Integrated Circuits." Electronics Magazine 19 Apr. 1965.
8. Bourgonjon, R. "The Evolution of Embedded Software in Consumer Products." International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, FL, 1995 (unpublished keynote address).

Note

1. On a multiprocessor system, the implementation is a bit more complex, but the programming and static analysis abstractions are comparable. The safety-critical Java profile is designed to support straightforward migration of hard real-time code from single-processor to multi-processor implementations.

About the Author

Kelvin Nilsen, Ph.D., is chief technology officer of Aonix, an international supplier of mission- and safety-critical software solutions. Nilsen oversees the design and implementation of the PERC real-time Java virtual machine along with other Aonix products, including ObjectAda compilers, development environment, libraries, and commercial off-the-shelf safety certification support. Nilsen's seminal research on the topic of real-time Java led to the founding of NewMonics, a leader in advanced real-time virtual machine technologies to support real-time execution of Java programs. In 2003, Aonix acquired NewMonics. Nilsen has a Bachelor of Science in physics from Brigham Young University and a Master of Science and doctorate degree both in computer science from the University of Arizona.

Aonix
877 S Alvernon WAY
Tucson, AZ 85711
Phone: (520) 991-6727
Fax: (520) 323-9014
E-mail: kelvin@aonix.com